Polychrony for refinement-based design

Jean-Pierre Talpin[1], Paul Le Guernic[1], Sandeep Kumar Shukla[2], Rajesh Gupta[3], Frédéric Doucet[3]
[1] INRIA/IRISA, [2] Virgina Tech, [3] UC San Diego, [4] UC Irvine

Abstract

The polychronous (i.e. multi-clocked) model of the SIGNAL *design language offers formal support for the capture of behavioral abstractions for both very high-level system descriptions (e.g.* SYSTEMC/SPECC*) and behavioral-level* IP *components (e.g.* VHDL*). Its platform,* POLYCHRONY*, provides models and methods for a rapid, refinement-based, integration and a formal conformance-checking of* GALS *hardware/software architectures. Its companion desynchronization techniques allow to model high-level, functional, specifications and formally verify successive design refinements yielding a distributed implementation. The present article demonstrates the effectiveness of this approach in refinement-based design by the experimental, comparative, case study of an even-parity checker design in* SPECC*. It highlights the benefits of the formal models, methods and tools provided in* POLYCHRONY*, in representing functional, architectural, communication and implementation abstractions of the design, and the successive refinements.*

1 Introduction

Rising complexities and performances of integrated circuits and systems, shortening time-to-market demands for electronic equipments, growing installed bases of intellectual property, requirements for adapting existing IPs with new services, all stress high-level design as a prominent research topics and call for the development of appropriate methodological solutions. In this aim, system design based on the so-called "synchronous hypothesis" consists of abstracting the non-functional implementation details of a system away and let one benefit from a focused reasoning on the logics behind the instants at which the system functionalities should be secured. From this point of view, synchronous design models [9] and languages [3] provide intuitive models for integrated circuits. This affinity explains the ease of generating synchronous circuits and verify their functionalities using compilers and related tools that implement this approach. The relational model of the SIGNAL/POLYCHRONY design language/plateform [9, 12] goes beyond the domain of purely synchronous circuits to embrace the context of architectures consisting of synchronous circuits and desynchronization protocols: GALS architectures. The unique features of this model are to provide the notion of *polychrony*: the capability to describe multi-clocked (or partially clocked) circuits and systems; and to support formal design *refinement*, from the early stages of requirements specification, to the later stages of synthesis and deployment, and by using formal verification techniques. In practice, a multi-clocked system description is often the representation or the abstraction of an asynchronous system or of a GALS architecture. In system-level design, the asynchronous implementation of a system is obtained through the refinement of its description towards hardware-software co-design. However, clocks are often left unspecified at the functional level, and no choice on a master clock made at the architectural level. As communication and implementation layers are reached, however, multiple clocks might be a way of life. In the polychronous design paradigm, one can actually design a system with partially ordered clocks and refine it to obtain master-clocked components integrated within a multiply-clocked architectures, while preserving the functional properties of the original high-level design, thanks to the formal verification methodology provided by the formal theory (model and theorems) of polychronous signals. In the present article, we put the principles of polychronous design to work in the context of the emerging high-level languages such as SYSTEMC/SPECC [7, 13, 14] by studying the refinement of a high-level specification, the even-parity checker (EPC) towards its implementation. Our goal is to derive conditions on specifications under which our design principles works. In other words, we seek towards tools and methodologies to allow to take a high-level SYSTEMC/SPECC specification and to refine it in a semantic-preserving manner into a GALS implementation. We focus on a simple

case study to illustrate our methodology and we show how the specification of the EPC in SPECC can be refined towards a GALS implementation with the help of POLYCHRONY.

2 An informal introduction to SIGNAL

In SIGNAL (figure 1), a process P consists of the composition of simultaneous equations over signals. A signal $x \in \mathcal{X}$ describes a possibly infinite flow of discretely-timed values $v \in \mathcal{V}$. An equation $\boldsymbol{x} = f\boldsymbol{y}$ denotes a relation between a sequence of operands $\boldsymbol{y}$ and a sequence of results $\boldsymbol{x}$ by a process $f \in F$. Synchronous composition $P \mid Q$ consists of considering a simultaneous solution of the equations P and Q at any time. SIGNAL requires three primitive processes: pre, to reference the previous value of a signal in time; when, to sample a signal; and default, to deterministically merge two signals (and provides, e.g. negation not, equality eq, identity id, etc). The equation $x =$ pre $v\, y$ initially defines x by v and then by the previous value of y in time (tags t_1, t_2, t_3 denote instants). The equation $x = y$ when z defines x by y when z is true. The equation $x = y$ default z defines x by y when y is present and by z otherwise. We exemplify the equational/relational design model of SIGNAL by considering the definition of a counting process: Count. It accepts an input event reset and delivers the integer output val. A local counter, initialized to 0, stores the previous value of val (equation counter := val$1 init 0). When the event reset occurs, val is reset to 0 (i.e. (0 when reset). Otherwise, counter is incremented (i.e. (counter + 1)). The activity of Count is governed by the clock of its output val, which differs from that of its input reset: Count is multi-clocked.

```
process Count = (? event reset ! integer val)
   (|  counter := val$1 init 0
    |  val     := (0 when reset)
                  default (counter + 1)
    |) where integer counter;
end;
```

3 A model of polychronous signals

Starting from the model of tagged signals of Lee et al. [8, 4], we give the tagged model of polychronous signals [9] for the formal study of protocol properties. We consider a set of boolean and integer *values* $v \in \mathcal{V}$ to represent the operands and results of a computations. A tag $t \in \mathbb{T}$, denotes an instant. The dense set is equipped with a *partial order* relation $\leq$ to denote synchronization and causal relations. The subset $\mathcal{T} \subset \mathbb{T}$ of a given process is chosen to be a semi-lattice $(\mathcal{T}, \leq, 0)$. A *chain* $C \in \mathcal{C}$ is a totally ordered subset of $\mathbb{T}$. An *event* $e \in \mathcal{E} = \mathcal{T} \times \mathcal{V}$ relates a tag and a value. A *signal* $s \in \mathcal{S} = \mathcal{T} \rightharpoonup \mathcal{V}$ is a partial function relating a *chain* of tags to a set of values. We write tags(s) for the domain of s. A *behavior* $b \in \mathcal{B} = \mathcal{X} \rightharpoonup \mathcal{S}$ is a partial function from signal names $x \in \mathcal{X}$ to signals $s \in \mathcal{S}$. We write vars(b) for the domain of b and $\text{tags}(b) = \cup_{x \in \text{vars}(b)} \text{tags}(b(x))$ for its tags. Hence, the informal sentence "x is present at t in b" is formally defined by $t \in \text{tags}(b(x))$. We write $b|_X$ for the projection of a behavior b on a set $X \subset \mathcal{X}$ of names (i.e. $\text{vars}(b|_X) = X$ and $\forall x \in X, b|_X(x) = b(x)$) and $b_{/X}$ for its complementary of $b|_{\text{vars}(b) \setminus X}$. A *process* $p \in \mathcal{P} = \mathcal{P}(\mathcal{B})$ is a set of behaviors that have the same domain X (written vars(p)). Synchronous composition $p \| q$ is defined by the set of behaviors that extend a behavior $b \in p$ by the restriction $c_{/\text{vars}(p)}$ of a behavior $c \in q$ if the projections of b and c on $\text{vars}(p) \cap \text{vars}(q)$ are equal.

$$p \| q = \left\{ \begin{array}{c} b \uplus c_{/\text{vars}(p)} \mid (b, c) \in p \times q, \\ b|_{\text{vars}(p) \cap \text{vars}(q)} = c|_{\text{vars}(p) \cap \text{vars}(q)} \end{array} \right\}$$

Scalability is a key concept for engineering systems and reusing components in a smooth design process. A formal support for allowing time scalability in design is given in our model by the so-called *stretch-closure property*. The intuition behind this relation is to consider a signal as an elastic with ordered marks on it (tags). If it is stretch, marks remain in the same order but have more space (time) between each other. If it is un-stretched, marks appear closer from each other and some are no longer observable. The same holds for a set of elastics: a behavior. If elastics are equally stretched, the partial order between marks is unchanged. A behavior c is a *stretching* of b, written $b \leq c$, iff $\text{vars}(b) = \text{vars}(c)$ and there exists a function $f : \mathcal{T} \to \mathcal{T}$ that - 1/ is strictly increasing - 2/ is monotonic along all chains - 3 satisfies $\text{tags}(c(x)) = f(\text{tags}(b(x)))$ for all $x \in \text{vars}(b)$, and $b(x)(t) = c(x)(f(t))$ for all $x \in \text{vars}(b)$ and all $t \in \text{tags}(b(x))$. Stretching is a partial-order relation. It gives rise to an equivalence relation between behaviors. The behaviors b and c are *stretch-equivalent*, written $b \lesseqgtr c$, iff there exists a behavior d s.t. $d \leq b$ and $d \leq c$. Both relations extend to processes. A process p is *stretch-closed* iff for all $b \in p$, $c \lesseqgtr b \Rightarrow c \in p$. A non-empty, stretch-closed process p admits a set of strict behaviors, written $(p)_{\lesseqgtr}$, s.t. $(p)_{\lesseqgtr} \subset p$ (for all $b \in p$, there is a unique $c \in (p)_{\lesseqgtr}$ s.t. $c \lesseqgtr b$).

Distributed design To model asynchrony, we consider a weaker relation which discards synchronization relations and allows for comparing behaviors w.r.t. the

$$P ::= \boldsymbol{x} = f\,\boldsymbol{y} \mid P \mid Q \mid P/x \quad f \in F \supseteq \{\,\mathsf{pre}\,v \mid v \in \mathcal{V}\} \cup \{\,\mathsf{when}\,,\,\mathsf{default}\,,\,\mathsf{not}\,,\,\mathsf{eq}\,,\,\mathsf{id}\,,\,\ldots\}$$

$$\begin{array}{rlll}
y: & (t_1,v_1)\,(t_2,v_2)\,(t_3,v_3)\ldots & y: & (t_1,v_1)\,(t_2,v_2)\,(t_3,v_3)\quad\ldots \\
\mathsf{pre}\ v\ y: & (t_1,v)\;\;(t_2,v_1)\,(t_3,v_2)\ldots & z: & (t_2,tt)\;\;(t_3,ff)\,(t_4,tt)\ldots \\
 & & y\ \mathsf{when}\ z: & (t_2,v_2)\quad\ldots
\end{array}$$

$$\begin{array}{rll}
y: & (t_2,v_2)\,(t_3,v_3)\ \ldots \\
z: & (t_1,v_1)\qquad (t_3,w_3)\ldots \\
y\ \mathsf{default}\ z: & (t_1,v_1)\,(t_2,v_2)\,(t_3,v_3)\ \ldots
\end{array}$$

Figure 1. Core-SIGNAL

sequences of values signals hold. The *relaxation* relation allows to individually stretch the signals of a behavior. A behavior c is a *relaxation* of b, written $b \sqsubseteq c$, iff $\mathrm{vars}(b) = \mathrm{vars}(c)$ and for all $x \in \mathrm{vars}(b)$, $b_{|x} \leq c_{|x}$. Relaxation is a partial-order relation that defines the flow-equivalence relation. Two behaviors are flow-equivalent iff their signals hold the same values in the same order. The behaviors b and c are *flow-equivalent*, written $b \approx c$, iff there exists a behavior d s.t. $d \sqsubseteq b$ and $d \sqsubseteq c$. The equivalence class of a behavior b is a semi-lattice: it admits a strict behavior, written $(b)_{\approx}$. We use relaxation to define the meaning of asynchronous composition $p \parallel q$ (we note $X = \mathrm{vars}(P)$, $Y = \mathrm{vars}(Q)$ and $I = X \cap Y$).

$$p \parallel q = \left\{ d \;\middle|\; \begin{array}{l} \exists b \in p, d_{|X\setminus Y} \lessapprox b_{|X\setminus Y},\ b|_I \sqsubseteq d|_I \\ \exists c \in q, d_{|Y\setminus X} \lessapprox c_{|Y\setminus X},\ c|_I \sqsubseteq d|_I \end{array} \right\}$$

Polychronous design properties The model of polychronous signals allows to define formal properties that are essential for the component-based design of GALS architectures. *Endochrony* is a key design property design. A process is endochronous iff, given an external (asynchronous) stimulation of its inputs I, it reconstructs a unique synchronous behavior (up to stretch-equivalence). Endochrony denotes the class of processes that are insensitive to (internal and) external propagation delays. A process p is endochronous on its input signals I iff $\forall b, c \in p$, $(b|_I)_{\approx} = (c|_I)_{\approx} \Rightarrow b \lessapprox c$. *Flow-equivalence* offers the right metric for checking the refinement of a high-level system specification with distributed communication protocols correct. For instance, it is considered in [2] for the refinement-based design of the LTTA protocol in SIGNAL. *Flow-invariance* is the property that ensures that the refinement of a functional specification $p|q$ by an asynchronous implementation $p \parallel q$ preserves flow-equivalence. Formally, p and q are flow-invariant iff, for all $b \in p|q$, for all $c \in p \parallel q$, $(b|_I)_{\approx} = (c|_I)_{\approx}$ implies $b \approx c$ for I the inputs of $p|q$. In SIGNAL, GALS architectures are modeled as *endo-isochronously* communicating endochronous components. We say that two endochronous processes p and q are endo-isochronous iff $(p|_I) \parallel (q|_I)$ is endochronous (with $I = \mathrm{vars}(p) \cap \mathrm{vars}(q)$). Endo-isochrony implies flow-invariance.

Capturing high-level design with polychrony In the polychronous design paradigm, one can give a functional-level specification of a system in terms of relations and partially-ordered clocks. A refinement, at the architecture-level, consists of isolating the master-clock of components and of integrating them within a multi-clocked architectures, while preserving the functional properties of the original design, thanks to the formal verification of flow-invariance. The main benefit of considering the model of polychronous signals for high-level C-like design languages lies in the formal semantics backbone/platform it provides, on which verification and optimization techniques can then be plugged in. There are several ways to envisage applying the POLYCHRONY model to high-level GALS architectures modeling in C-like design languages. A validation-based approach consists of associating every component to a description of its invariants (a behavioral type) and then to automate test-case generation to validate them. By contrast, a program analysis approach consists of automatically synthesizing this behavioral type and of representing it in an algebra in which deciding properties about these types (equivalence, bisimulation, etc) is decidable. Hence, our approach: abstract a component by a temporal formula or a SIGNAL process.

4 A case study: the even parity checker

The polychronous model of the SIGNAL design language offers formal support for the capture of behavioral abstractions for both very high-level system descriptions (e.g. SYSTEMC/SPECC) and behavioral-level IP components (e.g. VHDL). Its platform, POLYCHRONY, provides formal methods for a rapid, refinement-based, integration and a formal conformance-checking of GALS hardware/software architectures. The model of polychrony being quite elaborate, we focus on a case study that illustrate our methodology by showing how the specification of the EPC in SPECC can be refined towards a GALS implementation with the help of the tool POLYCHRONY, showing in what respects and at which critical design stages formal

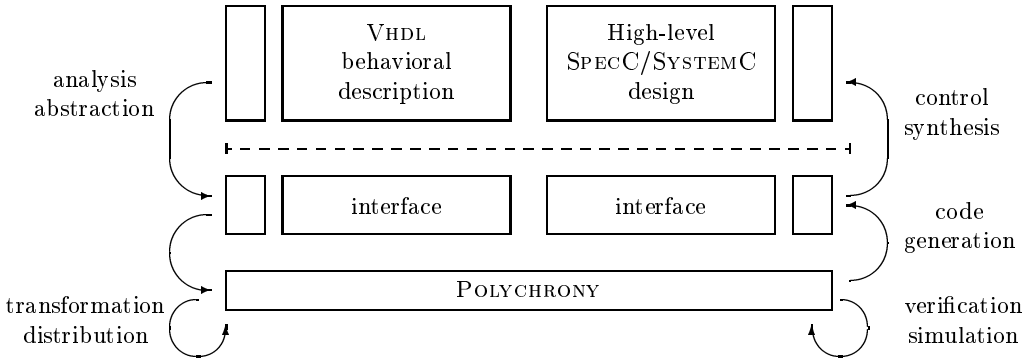
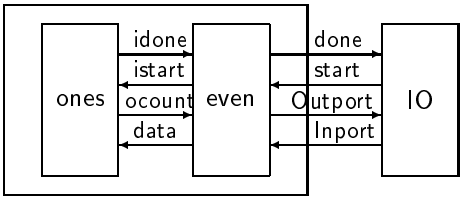


Figure 2. A polychronous component-based design platform

methods matter for engineering such architectures.



The Epc consists of three functional units: an IO interface process, an even test process and a main ones counting process. The behavior ones determines the parity of an input data received along Inport. Upon receipt of the start notification, it repeatedly shifts the data until it is 0ed. The output count ocount is sent along Outport and done notified.

```
behavior ones(...) {
  void main(void) {
    unsigned int data, ocount, mask, temp;
    while (1) { wait(start);
                data = Inport;
                ocount = 0;
                mask = 1;
                while(data != 0){temp = data & mask;
                                 ocount += temp;
                                 data >>= 1;
                }
                Outport = ocount;
                notify(done);
    }}
};
```

The translation of the behavior ones in Signal consist, first, of decomposing the syntactic structure of the SpecC program into an intermediate representation that renders the imperative structure of the original program together with its most characteristic features (use of locks, interrupts, etc). In this structure, each thread consists of a sequence of blocks (critical sections) delimited by wait and notify synchronization statements. Within such blocks, basic control structures are then encoded. A method call or a basic operation, e.g. $x = y + 1$, is encoded by an equation, e.g. either $x = y\$1 + 1 \,\mathsf{when}\, c$ (when y references a value computed during the previous transition in this block) or $x = y + 1 \,\mathsf{when}\, c$ (if it has already been computed in the same transition), conditioned by an activation clock c. A conditional statement, e.g. $\mathsf{if}\, x \,\mathsf{then}\, P \,\mathsf{else}\, Q$, is encoded by constraining the clock of P by x and that of Q by $\mathsf{not}\, x$. Internal while loops are encoded by over-sampling. Interrupts are rendered by boolean signals that tell whether or not they are raised during a computation. An interrupt conditions the activation clock of subsequent equations in the control flow graph; if it escapes the scope of the method in which it is raised, it becomes an output signal of the process that encodes the method in order to propagate in the context of use of that method. The encoding of the behavior ones in Signal yields a process in which the clock of input/output signals are synchronized to input/output notification events. The process ones consists of one critical section. The internal while loop is encoded by an over-sampling sub-process.

```
process ones = (? integer Inport; event start
                ! integer Outport; event done)
  (| start   ^= Inport
   | Outport := ocount when data=0
   | data    := Inport default rshift(data$1 init 0xffff)
   | ocount  := (ocount$1 init 0) + xand(data, 1)
   | done    ^= Outport
   |) where integer data, ocount end;
```

Architecture layer design refinement The encoding of the event-parity checker demonstrates the capability of Signal to give a synchronous model of components for specification-level SpecC designs. This level of abstraction (synchrony) allows for decoupling the specification of the system under design from (too) early architecture mapping choices. It additionally allows for an optimized recombination of behaviors (e.g. the Signal compiler could for instance be used to merge the other IO and even behaviors into a single SpecC fsm, using clock hierarchization techniques [1]

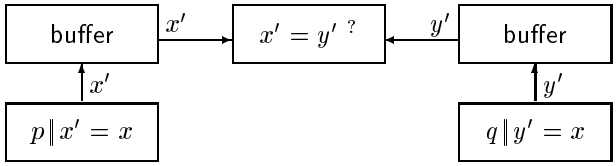
(i.e. the core engine of the SIGNAL compiler). Suppose we have done so and consider the architecture layer of the SPECC even-parity checker example. We now have two behaviors, ones and even+io that communicate asynchronously via the ChMP channel.

```
channel ChMP() implements iSend, iRecv {
    unsigned int data; event eReady, eAck;
    bool ready_flag = false, ack_flag = false;
    void send(unsigned int v)  {
        data = v;
        ready_flag = true;
        notify(eReady);
        while(!ack_flag) wait(eAck);
        ready_flag = false;
        notify(eReady);
        while (ack_flag) wait(eAck);
    }
    ⋮
}};
```

Modeling the architecture layer in SIGNAL requires an abstraction of the virtual simulation kernel semantics for the wait/notify statements. This kernel can be simulated in a way similar to the SIGNAL library that implements the ARINC 653 specification [6]. It arbitrates the suspension (by a wait statement) and resumption (by a notify) of scheduled processes by inhibiting/releasing their main clock (clk in the example). The send/receive wrappers use a lock eReady and boolean flags to prevent concurrent accesses to the shared data. Focus on method send (receive is its dual). It consists of two critical sections that end with a wait statement. The SIGNAL translation renders each section by a sub-process whose activity is governed by an event (s0 or s1).

```
process send = (? unsigned v, event clk ! )
  (| % controller
     s :=         (0 when (s$1=1) ^* (not ack_flag) ^* clk)
          default (1 when (s$1=0) ^* (ack_flag) ^* clk)
   | % critical section 1
     (| s0 := when s=0
      | data := v when s0
      | ready_flag := true when s0
      | notify(eReady when s0)
      | wait(eAck when (s0 ^* (not ack_flag)))
      |)
   | % critical section 2
     (| s1 := when clk
      | ready_flag := true when s1
      | notify(eReady when s1)
      | wait(eAck when (s1 ^* ack_flag))
      |)
   |) where event s0, s1; integer s init 1;
```

Showing that the refinement of the EPC architecture layer preserves flow-equivalence w.r.t. the specification level amounts to a model checking problem (implemented by using, e.g., the tool SIGALI). Its overall principle is illustrated below. Consider two processes p and q sharing a signal x. Showing p's x and q's x flow-equivalent can be implemented by installing an observer connected to p and q by a one-place buffer of a FIFO queue. The observer repeatedly checks whether its copy x'' of the nth. value of p matches the copy y'' of the nth. value of q. Verifying p and q flow-invariant amounts to checking that the value of the observer is invariantly true.



Communication and RTL layers The communication layer of the EPC mainly consists of a data-type refinement of the ChMP channel and of the decomposition of the renamed methods send and receive into sub-procedures. It intends to make the implementation of the ChMP as a bus explicit.

```
channel cBus() implements iBus {
    ⋮
    void write(unsigned bit[31:0] wdata) {
        ready.assign(1);
        data = wdata;
        ack.waitval(1);
        ready.assign(0);
        ack.waitval(0);
}};
```

The RTL layer of the EPC consists of the introduction of a master clock clk and of a reset signal rst together with the convertion of the EPC communication-layer specification into finite-state machine code. This translation closely corresponds the SIGNAL's encoding of the EPC into blocks (critical sections). Checking the RTL-level refinement correct amounts to proving it bisimilar to the encoding of the communication-layer, i.e., showing that flow-equivalence along the input/output signals istart, idone, data, ocount is preserved.

```
behavior ones(...)  {   void main(void) {
    unsigned bit[31:0] data, ocount, mask, temp;
    enum state {S0, S1, S2, S3, S4, S5, S6, S7} state;
    state = S0;
    while (1) {
      wait(clk);
      if (rst == 1b) state = S0;
      switch (state) {
        case S0: done = 0b;
                 ack_istart = 0b;
                 if (start == 1b) state=S1
                             else state=S0;
                 break;
        case S1: ack_istart = 1b;
                 data = inport;
                 state = S2;
                 break;
```

```
        case S2: ocount = 0;
                 state = S3;
                 break;
        case S3: mask = 1;
                 state = S4;
                 break;
        case S4: temp = data & mask;
                 state = S5;
                 break;
        case S5: ocount = ocount + temp;
                 state = S6;
                 break;
        case S6: data = data >> 1;
                 if (data == 0) state=S7
                           else state=S4;
                 break;
        case S7: outport=ocount;
                 done = 1b;
                 if (ack_idone == 1b) state=S0
                                 else state=S7;
                 break;
}}}};
```

Toward an integration platform In the aim of automating the above process within a versatile component integration platform, the use of SIGALI [11] as a refinement (model) checking tool directly provides the required support for automating this process by using controller synthesis techniques [10]. Whereas model-checking consists of proving a property correct w.r.t. the specification of a system, controller synthesis consists of using this property as a control objective and to automatically generate a coercive process that wraps the initial specification so as to guarantee that the objective is an invariant.

5 Conclusion

We have put a polychronous design model to work for the refinement of a high-level even-parity checker in SPECC from the early stages of its functional specification to the late stages of its hardware/software GALS implementation. We demonstrated the effectiveness of this approach by showing in what respects and at which critical design refinement stages formal verification and validation support was needed, highlighting the benefits of using the tool POLYCHRONY in that design chain. The novelty of integrating POLYCHRONY in a high-level design tool-chain lies in the formal support offered by the former to automate critical and complex design verification and validation stages yielding a correct-by-construction system design and refinement in the latter.

References

[1] AMAGBEGNON, T. P., BESNARD, L., LE GUERNIC, P. "Implementation of the data-flow synchronous language SIGNAL". In *Conference on Programming Language Design and Implementation*. ACM Press, 1995.

[2] BENVENISTE, A., CASPI, P., LE GUERNIC, P., MARCHAND, H., TALPIN, J.-P., TRIPAKIS, S. "A protocol for loosely time-triggered architectures". In *Embedded Software Conference*. Springer Verlag, October 2002.

[3] BERRY, G., GONTHIER, G. "The ESTEREL synchronous programming language: design, semantics, implementation". In *Science of Computer Programming*, v. 19, 1992.

[4] CARLONI, L. P., MCMILLAN, K. L., SANGIOVANNI-VINCENTELLI, A. L. "Latency-Insensitive Protocols". In *Proceedings of the 11th. International Conference on Computer-Aided Verification*. Lecture notes in computer science v. 1633. Springer Verlag, July 1999.

[5] F. DOUCET, M. OTSUKA, R. GUPTA AND S. SHUKLA "Efficient System Level Co-Design Environment using Split Level Programming". Technical Report 01-34, CECS/UCI, June 2001.

[6] A. GAMATI, T. GAUTIER "Modeling of modular avionics architectures using the synchronous language SIGNAL". In *proceedings of the Euromicro conference on real-tile systems*. June 2002.

[7] D. GAJSKI, F. VAHID, S. NARAYAN, AND J. GONG. "Specification and Design of Embedded Systems". Prentice Hall, 1994.

[8] LEE, E. A., SANGIOVANNI-VINCENTELLI, A. "A framework for comparing models of computation". In IEEE *transactions on computer-aided design*, v. 17, n. 12. IEEE Press, December 1998.

[9] LE GUERNIC, P., TALPIN, J.-P., LE LANN, J.-L. Polychrony for system design. In *Journal of Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design*. World Scientific, 2002.

[10] MARCHAND, H., BOURNAI, P., LE BORGNE, M., LE GUERNIC, P. Synthesis of Discrete-Event Controllers based on the Signal Environment. In *Discrete Event Dynamic System: Theory and Applications*, v. 10(4), pp. 325–346, 2000.

[11] H. MARCHAND, E. RUTTEN, M. LE BORGNE, M. SAMAAN. Formal Verification of SIGNAL programs: Application to a Power Transformer Station Controller. *Science of Computer Programming*, v. 41(1), pp. 85–104, 2001.

[12] ESPRESSO project. `http://www.irisa.fr/espresso`

[13] SPECC. `http://www.specc.org`

[14] SYSTEMC. `http://www.systemc.org`